An investigation of data reuse on the Cray S-MP system 500

A. Stewart, M. Louter-Nool, H.J.J. te Riele, D.T. Winter

Department of Numerical Mathematics

Report NM-R9415 July 1994

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

An Investigation of Data Reuse on the Cray S-MP System 500

A. Stewart, M. Louter-Nool, H.J.J. te Riele and D.T. Winter

CWI

P.O.Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

The Cray S-MP is a multiprocessor computer architecture; processors have a local data cache and have access to a main shared memory. In order to reduce the overheads associated with accessing main memory it is beneficial to maximise the reuse of data stored in cache. This report describes some data reuse refinements of a smoothing operation (used to obtain an approximate solution to an elliptic PDE) and the results of implementing them on the Cray S-MP.

1991 Mathematics Subject Classification: Primary: 69C12. Secondary: 65Y05, 65Y10
1991 CR Categories: C.1.2, G.1.3.
Keywords & Phrases: Memory Bottleneck, Data Reuse, Data Partitions, Synchronisation, Models for predicting execution performance.

Note: The work of the first author was supported by an ERCIM fellowship and his current address is The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

1. Introduction.

One characteristic of many multiprocessor computer architectures is non-uniform data access times; local (on chip) memory can be accessed very rapidly whereas non-local information can be very expensive to reference. Indeed non-local information may even have a hierarchical organisation and a hierarchical cost structure - see for example the Kendal Square [7]. It seems likely that the efficient utilisation of such machines is dependent on suitable algorithmic refinements (if any exist) which partition data structures over processors so that computations only involve local data references (or, at least, minimise non-local references).

The Cray S-MP [1] is a shared memory multiprocessor machine with non-uniform data access costs. Local cache memory can be accessed very rapidly whereas the transfer of data into and out of cache is relatively expensive. A 28 processor machine has been used to carry out a set of data reuse experiments. The architecture of the Cray S-MP is such that a processor is linked to main memory by a shared bus. In the case of a 28 processor machine there are 7 shared buses with 4 processors per bus.

Only one processor can access a bus at a given time; however, different buses may operate concurrently. Consequently, there are two overheads associated with non-local data references on the Cray S-MP:

(i) the relatively expensive operation of accessing non-local data; and

(ii) the possible delay of other processors sharing the same bus.

Report NM-R9415
ISSN 0169-0388
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

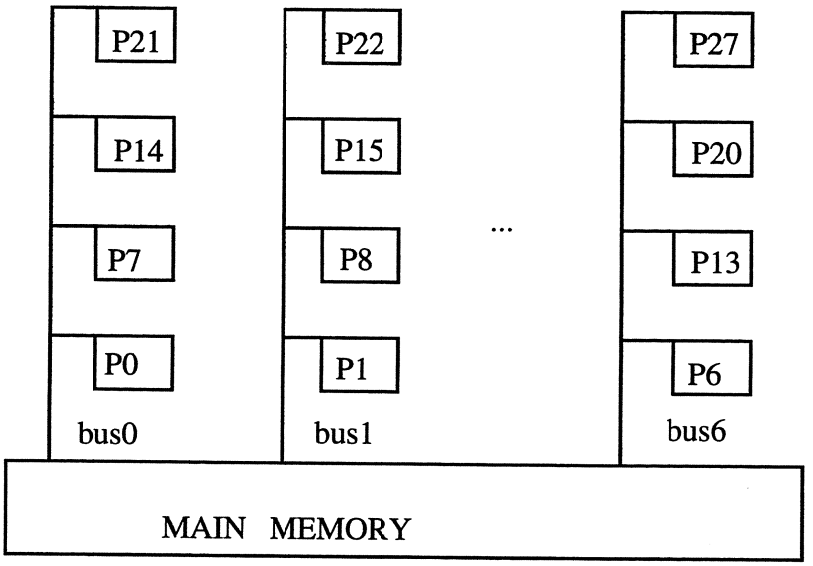

Figure 1. Architecture of CRAY S-MP.

In the worst case, when the traffic on the buses is very heavy, the performance of the machine will deteriorate to that of a one processor per bus machine or even worse.

The remainder of this report is organised as follows: a basic smoothing method suitable for the solution of elliptic PDEs is described in Section 2. Two methods of partitioning the data grids associated with the smoothing operation are discussed in Section 3. A predictive performance model of the Cray S-MP is derived in Section 4. The actual and predicted performances of two algorithmic refinements are compared in Section 5. A method for further increasing data reuse, operation composition, is proposed in Section 6. The basic feature of this refinement is that synchronisation points can be removed at the cost of extra computation. Finally, some conclusions are presented in Section 7.

2. A Basic Smoothing Algorithm

An approximate solution to an elliptic PDE may be generated by the Red Black Gauss-Seidel method [2]. Consider the equation

$$\Delta^2 \mathbf{u} = \mathbf{f} \tag{2.1}$$

on a 2D plane section with $\mathbf{u}$ given on the boundaries. Suppose that a discrete grid $\{ (0+ih , 0+jh) \mid 0 \le i \le N \ \& \ 0 \le j \le N \}$ of step length h is used to generate an approximate solution to $\mathbf{u}$. A method for improving an approximate solution $\mathbf{u}_{i,j}$ is:

$$\mathbf{u}_{i,j} = (\mathbf{u}_{i+1,j} + \mathbf{u}_{i-1,j} + \mathbf{u}_{i,j+1} + \mathbf{u}_{i,j-1} - h^2\mathbf{f}_{i,j}) / 4 \tag{2.2}$$

The calculations of $\mathbf{u}_{i,j}$ and $\mathbf{u}_{i+1,j}$ in parallel may cause mutual interference. In order to prevent such interference the grid may be subdivided into a set of *red* points { $(0+ih, 0+jh) \mid \text{even}(i+j)$ } and a set of *black* points { $(0+ih, 0+jh) \mid \text{odd}(i+j)$ }. A calculation at a red (black) point cannot cause interference with a calculation at any other red (black) point - hence, the calculations (2.2) may be carried out in parallel for all red (black) points. The Red Black Gauss-Seidel method consists of applying (2.2) to calculate new approximate solutions at the set of red points and subsequently applying (2.2) to generate new approximations at the set of black points.

3 . Organisation of Data in Cache.

One factor which significantly influences the performance of the Cray S-MP is the strategy used to store data in cache. For example, a processor may be assigned the task of calculating new approximations (2.2) for a set of consecutive red points in a *vector*: $\mathbf{u}_{i,j_c}$, $i_l \le i \le i_u$, odd i and j_c an odd constant. The calculation may be implemented as follows:

LOAD $\mathbf{u}_{i,j_c}$	$i_l - 1 \le i \le i_u + 1$	& even i ;	(black)	
LOAD $\mathbf{u}_{i,j_c+1}$	$i_l \le i \le i_u$	& odd i ;	(black)	
LOAD $\mathbf{u}_{i,j_c-1}$	$i_l \le i \le i_u$	& odd i ;	(black)	
LOAD $\mathbf{f}_{i,j_c}$	$i_l \le i \le i_u$	& odd i ;	(red)	(3.1)
CALCULATE $\mathbf{u}_{i,j_c}$	$i_l \le i \le i_u$	& odd i ;	(red)	
STORE $\mathbf{u}_{i,j_c}$	$i_l \le i \le i_u$	& odd i ;	(red)	

However, only the first data vector is being reused in the calculation of (2.2) - i.e. the first data vector is used as input for two arithmetic operations whereas all of the remaining data is only associated with a single arithmetic operation.

Data reuse within cache can be improved by assigning a processor the task of calculating new approximations (2.2) for the set of red points within a given *block*: $\mathbf{u}_{i,j}$ $i_l \le i \le i_u$ & $j_l \le j \le j_u$ & even $(i+j)$. In this case the operation can be implemented as follows:

LOAD $\mathbf{u}_{i,j}$	$i_l - 1 \le i \le i_u + 1$	& $j_l - 1 \le j \le j_u + 1$	& odd $(i+j)$;	(black)	
LOAD $\mathbf{f}_{i,j}$	$i_l \le i \le i_u$	& $j_l \le j \le j_u$	& even $(i+j)$;	(red)	(3.2)
CALCULATE $\mathbf{u}_{i,j}$	$i_l \le i \le i_u$	& $j_l \le j \le j_u$	& even $(i+j)$;	(red)	
STORE $\mathbf{u}_{i,j}$	$i_l \le i \le i_u$	& $j_l \le j \le j_u$	& even $(i+j)$;	(red)	

Now each interior element of the first data block can be used as input to 4 arithmetic operations. It should be noted that the vector-wise data partition(3.1) can be modified so that input data is reused in 4 operations; this can be accomplished by calculating the vector $\mathbf{u}_{i,j_c+1}$, $i_l \le i \le i_u$, even(i) immediately after $\mathbf{u}_{i,j_c}$ and reusing the input vectors $\mathbf{u}_{i,j_c}$ and $\mathbf{u}_{i,j_c+1}$. These examples illustrate the concept of data reuse. However, it is

necessary to describe the architectural structure of the Cray S-MP before considering in detail the best data reuse implementation strategies.

4. Architectural Details of the Cray S-MP.

The Cray S-MP has the potential to operate in parallel at several levels from the processor level down to sub-processor levels. Consequently, it can be complex to achieve a high degree of efficiency. Each Intel i860 processor [6] has two arithmetic units, addition and multiplication, which can operate concurrently. Furthermore both of these units are built from a number of component stages and therefore have the potential to operate in either scalar or pipelined (vector) mode. A processor can perform a DOUBLE PRECISION scalar addition in 75 nanoseconds and a DOUBLE PRECISION multiplication in 100 nanoseconds. The units can only operate in pipelined mode if the initial elements of the input data are stored on **quad word boundaries** in cache. A word in the wth position in cache is on a quad word boundary if w MOD 4 = 1. The quad word boundary restriction applies to pipelined loading and storing operations as well as to pipelined arithmetic operations, when coded in a high order language. The peak performances of the units are 25 nanoseconds for DOUBLE PRECISION addition and 50 nanoseconds for DOUBLE PRECISION multiplication. However, the actual performance of the units depends critically on the input vector lengths. Experimental results estimate $n_{1/2}$ for multiplication to be 14 where $n_{1/2}$ is the vector length needed to achieve one half of peak performance for the unit [4]. The average time per multiplication, t_e, can be determined for a vector length n and a peak performance t_p by [4] [5]:

$$t_e = t_p + (n_{1/2}\, t_p) / n. \tag{4.1}$$

$n_{1/2}$ is related to the pipeline start up time t_0 by

$$n_{1/2} = t_0 / t_e. \tag{4.2}$$

An estimation of the vector lengths that are needed to achieve a loss in efficiency for pipelined multiplication of no more than 20% and 10% are 70 and 140, respectively. It should be noted that $n_{1/2}$ is a constant of the *hardware*; however, the approximation to $n_{1/2}$ has been derived from *software* experiments and, therefore, is an over-estimation.

The size of vectors that can be passed as input to the functional units depends on the size of the data cache - in the case of the Cray S-MP each processor has an 8K byte data cache (i.e. 1024 DOUBLE PRECISION reals). The peak arithmetic speeds may be further degraded by the time taken to load data into and out of cache [5]. There is a significant latency associated with a processor acquiring exclusive use of a bus. The peak transfer rate of a DOUBLE PRECISION data item to/from cache is 50 nanoseconds. However, there is again an initial communication latency; experimental results estimate $n_{1/2}$ for loading DOUBLE PRECISION reals to be 50.

The addressing mechanism of the hardware is indirect and can result in further degradation of performance. Virtual addresses are generated by the compiler and these must be converted to actual addresses at run time. A processor is loaded with a fast but incomplete conversion table; if a virtual address does not occur in this table (a TLB miss) then it is necessary to search a "slow" table in main memory. TLB misses result in increased bus use and deterioration of performance. This problem can be negligible for **long vector** implementations but can cause significant delays for accessing contiguous rows or subblocks of a matrix [1] [6].

The details of the hardware can be used to construct a predictive performance model of the Cray S-MP. For a given program it is possible to predict the sequential time to load data into cache, compute arithmetic operations on cache data and store data into main memory. If vector lengths are >> $n_{1/2}$ for loading then the peak performance rates are used to make predictions; otherwise the average performance rates (4.1) for a given operation and vector length are used. A predicted MFLOPS rate can be calculated by dividing the total number of operations that are applied to cache data by the predicted time. No allowances are made in the predictions for indirect addressing.

5. Two Implementations of a Smoothing Operation on the Cray S-MP.

Two factors which can significantly influence the performance of the S-MP are:

(i) the length of vector operands;

(ii) the amount of data reuse.

Two implementation refinements of smoothing are described in this section. The strategy of the first refinement is to maximise the length of vector operands whereas the strategy of the second refinement is to maximise data reuse. Data reuse will be quantified by

$$\textbf{leverage} = \frac{\textbf{time for arithmetic computation}}{\textbf{time for data transfer to cache}} . \qquad (5.1)$$

For a 28 processor machine (with 7 buses) a leverage of more than 3 is required in order to obtain the full benefit of having 4 processors per bus i.e. while three processors are computing on local data the fourth processor on the same bus can acquire access to main memory.

A model of the first refinement, referred to as **long vector** implementation, assumes that the calculation of a vector $\mathbf{u}_{i,j_C}$, $i_l \le i \le i_u$, odd i and j_C an odd constant, of red approximations is achieved by (2.2) by loading 4 vectors of black points $\mathbf{u}_{i+1,j_C}$, $\mathbf{u}_{i-1,j_C}$, $\mathbf{u}_{i,j_C+1}$, $\mathbf{u}_{i,j_C-1}$ and a (scaled by h^2) vector of right hand sides $\mathbf{f}_{i,j_C}$. A more detailed description of the implementation model can be found in Appendix B.

A performance prediction for the **long vector** implementation can be made by deriving a time for the calculation of a single approximation (in a vector) based on pipelined loading and arithmetic operation timings (cf. Section 4):

TIME = 5 LOADS + 1 STORE + 4 PIPELINED ADDITIONS + 1 PIPELINED MULTIPLICATION

$$=5 * 50 + 1 * 50 + 4 * 25 + 1 * 50 = 450 \text{ nanoseconds.} \quad (5.2)$$

Therefore, the time/operation is 90 nanoseconds; this is equivalent to 11.1 MFLOPS. The predicted leverage of the operation is 150/300 = 0.5. In other words, it is predicted that for this implementation the effective number of processors/shared bus is 1.5 (= leverage+1).

The actual performance results for the **long vector** refined smoothing operation are given in Table 1, where * denotes a decrease in performance. The effective number of processors/shared bus is 8.3/5.9 =1.4.

Processors	Time (sec)	MFLOPS	Speed Up
1	19.49	8.04	-
7	3.30	47.52	5.9
14	2.42	64.79	8.1
21	2.35	66.72	8.3
28	*	*	*

Table 1. Performance results of **long vector** scheme after 100 iterations on a 562 × 562 grid.

The second refinement is designed to increase leverage(5.1). To that end the block loading strategy described in Section 3 is applied. Cache is used to store two (12 × 12)-square blocks of black **u** values, two (10 × 10)-square blocks of red **f** values and store the results of calculating two (10 ×10)-blocks of red **u** values. The method is implemented using *scalar* arithmetic because of the limited number of cache library subroutines that are provided with the system (see Appendix C). The predicted time to calculate a red (10 × 10)-subblock is given by:

$$\begin{aligned} \text{TIME} = \; & (12 \times 12)\text{-LOAD} + (10 \times 10)\text{-LOAD} + \\ & (10 \times 10)\text{-CALCULATIONs (2.2)} + (10 \times 10)\text{-STORE.} \end{aligned} \quad (5.3)$$

Assuming peak loading performance, this is equivalent to 8.7 MFLOPS on a single processor with a leverage of 2.3. The predicted performance, assuming $n_{1/2}$ for loading to be 50, is 7.7 MFLOPS per processor with a leverage of 1.6. The performance results for the **data reuse** refined smoothing scheme are given in Table 2.

Processors	Time (sec)	MFLOPS	Speed Up
1	22.28	7.0	-
7	3.64	43.0	6.1
14	2.17	72.5	10.2
21	2.16	72.6	10.3
28	*	*	*

Table 2. Performance results of **data reuse** refined smoothing scheme after 100 iterations on a 562 × 562 grid.

It is interesting to note that although pipelined arithmetic should be 2-3 times faster than scalar arithmetic the first implementation is only 15% faster on a single processor than the second implementation on a single processor. This result confirms that the performance bottleneck for the Cray S-MP is memory access. It should also be noted that the two performance constraints of (i) leverage and (ii) vector length/alignment may be incompatible.

6. Operation Composition in Cache.

The Cray S-MP can achieve high performance when computing high intensity calculations - i.e. when the number of operations that can be applied is at least an order of magnitude more than the number of data items in cache. One example of high intensity computation is matrix-matrix multiplication [8]. However, the ratio of operations to cache data for the red and also the black phase of smoothing for the block case is only 5/2. For some such low intensity computations it may be possible to increase leverage by altering the synchronisation points of a program in order to reuse cache data that has been previously loaded or calculated.

The implementations of smoothing described in Section 5 are such that an operator (e.g. red phase smooth) is applied over a complete grid before another operator (e.g. black phase smooth) is applied - i.e. a synchronisation point is defined between the two operations. This means that red approximations are calculated in cache blocks in the first phase, stored in main memory and subsequently, as the cache will have been overwritten, reloaded back into cache in the second phase. Better use of memory can be made by applying the black phase operator when the calculated red approximations remain in cache - i.e. if the synchronisation point is removed then the operations could be combined and applied to cache data removing the need for additional reloads.

The purpose of the synchronisation point is to ensure that black values are computed from the *most recently* calculated red values. A computation carried out without such synchronisation may result in intermediate grid states which contain a mixture of recently computed subblocks and "old" subblocks. Therefore, any attempt to load a subblock of red values with a surrounding boundary layer could result in erroneous input. This difficulty can be overcome by performing additional computations in order to ensure that "new" red values are guaranteed to be present on the boundary layer. Consider the calculation of the red and black values for an $(N \times N)$-subblock. If the red values are calculated for an appropriate $(N+2) \times (N+2)$-subblock then an $(N \times N)$-block of black values can be subsequently computed without further data loads. However, the computation of an $(N \times N)$-block without intermediate synchronisation requires $2N$ additional red calculations. The effectiveness of the approach depends on the ratio $2N : N^2$ - i.e. the percentage of additional computation. If the cache is sufficiently large then the increased computational load can be offset by reduced memory access time and increased leverage.

Experiments have been carried out to evaluate the effectiveness of combining three operations for cache data:

SMOOTHING (RED PHASE) ;
SMOOTHING (BLACK PHASE) ;
RESIDUAL CALCULATIONS (RED POINTS ONLY).

The residual calculations (defined in Appendix D) are needed to determine convergence (and could also be used in the implementation of multigrid acceleration techniques [3]). An outline of the implementation of the **combined smoothing & residual** operation is described below:

LOAD $\{(N+6) \times (N+6)\}$-block	- black **u** points;
LOAD $\{(N+4) \times (N+4)\}$- block	- **f** points;
CONDITIONAL LOAD two $(N+4)$-vectors	- boundary red **u** vectors;
CALCULATE (2.2) for $\{(N+4) \times (N+4)\}$-block	- red **u** points;
CALCULATE (2.2) for $\{(N+2) \times (N+2)\}$-block	- black **u** points;
CALCULATE RESIDUALS for $(N \times N)$-block	- red **u** points;
STORE $(N \times N)$-block	- **u** points;

In this case the application of the three combined operations to an $(N \times N)$-cache block requires $4N$ additional red/black calculations. The implementation is described in more detail in Appendix D.

A performance prediction for this implementation using $n_{1/2}$ for loading to be 50 is 7.5 MFLOPS on a single processor with a leverage of 2.9. The prediction assumes the worst case leverage conditions - i.e. it is assumed that red boundary values are loaded. The performance results of the **combined smoothing & residual** operation are given in Table 3.

Processors	Time (sec)	MFLOPS	Speed Up
1	32.76	5.6	-
7	5.24	34.8	6.3
14	3.34	54.6	9.8
21	2.95	61.8	11.1
28	*	*	*

Table 3 Performance results for **combined smoothing and residual** scheme after 100 iterations on a 450×450 grid.

The disadvantage of this combined operation implementation is the relatively small data blocks that are transferred to/from cache. Matters can be improved in this regard by grouping the red and black points together in cache. This means that all of the points of a subblock (including unnecessary red points) must be transferred to cache. However, larger data blocks of size 20×20 can be processed in cache. Cache is organised into two blocks of size 20×20 with residuals being calculated for (14×14)-blocks. The predicted performance of this modified implementation is 7.6 MFLOPS with a leverage of 3.4 (see Appendix E). The performance results of the **modified combined smoothing & residual** operation are given in Table 4.

Processors	Time (sec)	MFLOPS	Speed Up
1	39.07	4.7	-
7	6.00	30.4	6.5
14	3.34	54.6	11.7
21	2.52	72.3	15.5
28	2.20	82.8	17.8

Table 4 Performance results for **modified combined smoothing and residual** scheme after 100 iterations on a 450×450 grid.

This implementation performs approximately 15% better than a corresponding **long vector** implementation of the same 3 operation sequence.

Finally, it should be noted that it is simpler to construct more efficient implementations of iterative methods for solving more complex PDEs. For example, a **long vector** version of a Red Black Newton approximation method for the solution of the equation

$$\Delta^2 \mathbf{u} + a\, e^{b\mathbf{u}} = \mathbf{f}$$

has been implemented and delivers a speed-up of 16 on 28 processors.

7. Conclusions.

It is well known that memory access is often the limiting performance feature of shared memory machines. Peak arithmetic performance can only be realised in very special circumstances when the number of arithmetic operations is an order of magnitude greater than the number of data accesses. For *general* programs the time to access main memory severely restricts the overall performance. For example, the Intel i860 processor can achieve (with overlapped DOUBLE PRECISION addition and multiplication) 40 - 60 MFLOPS. However, without data reuse and overlapped floating point operations the effective processor performance drops to 10 MFLOPS.

Memory access time is crucial not only for the sequential processor performance on the Cray S-MP but also for the parallel speed-up. If many main memory accesses are required by an application then the potential gains from having multiple processors on a shared bus are severely restricted. The experiments that have been carried out indicate that it is very difficult, for certain types of problems, to avoid unwanted bus contention. If the present ratio of memory access speed to arithmetic speed persists then the concept of improving performance, for *general* applications, by having multiple processors on a shared bus is limited in its scalability.

The individual processors of the Cray S-MP are complex entities. A number of the hardware characteristics are pertinent to the construction of a predictive performance model of the Cray S-MP: the vector half lengths of the pipelined operations, the quad word alignment restrictions and indirect addressing can have significant effects on execution speed. In general, it is necessary to use a complex performance

model in order to construct efficient programs for the Cray S-MP. Sadly, the use of complex performance models may result in complex and lengthy programs; consequently the reliability and robustness of such software is questionable.

It is crucial that a balance is found between the complexity of utilising hardware efficiently and the necessity for reasoning about programs in a simple and transparent way. At present there are a large number of distinct parallel machines and corresponding large number of machine dependent FORTRAN/C dialects. This intolerable situation results in scientific software having a very short life span - as long as it takes for a new faster parallel machine (and language) to become available. It is crucial for the development and construction of reliable scientific programs that a relatively simple abstract model of parallel computation is found so that a bridge [9] can be made between hardware and software.

Appendix A: Software Restrictions

The efficient transfer of data between main memory and cache is realised by calling specialised library subroutines. There are several software restrictions on the structure of data that can be transferred:

(i) data must be stored/read from *consecutive* locations of cache columns;

(ii) if the data is represented as a multi-dimensional array in main memory then the number of elements in each dimension that can be **block** transferred must be even;

(iii) the data to be transferred must have a constant stride in each dimension in main memory.

Appendix B: A model of the long vector implementation

Consider the calculation of the red vector $\mathbf{u}_{i,j_C}$, $i_l \le i \le i_u$, odd i and j_C an odd constant. As a first step it is necessary to load the black vectors $\mathbf{u}_{i+1,j_C}$, $\mathbf{u}_{i-1,j_C}$, $\mathbf{u}_{i,j_C+1}$, $\mathbf{u}_{i,j_C-1}$ and a (scaled by h^2) vector of right hand sides $\mathbf{f}_{i,j_C}$. Conventional optimisations could be based on the fact that the vectors $\mathbf{u}_{i+1,j_C}$ and $\mathbf{u}_{i-1,j_C}$ have many elements in common. However, in order to exploit the potential of the pipelined arithmetic units these input vectors should be quad word aligned. Vectors may only be loaded into *consecutive* locations in cache with the available software (see Appendix A). Consequently it is only possible to achieve a quad word alignment of the vectors $\mathbf{u}_{i-1,j_C}$ and $\mathbf{u}_{i+1,j_C}$ within a single cache column by loading $\mathbf{u}_{i,j_C}$ as well (an unnecessary load). This unnecessary load may be avoided by loading the vectors $\mathbf{u}_{i-1,j_C}$ and $\mathbf{u}_{i+1,j_C}$ separately. However, in this case, the fact that the vectors are overlapping can not be exploited. The model of the implementation assumes that $5n$ LOADs and n STOREs are required to calculate a result vector of length n.

The actual implementation of the **long vector** refinement is provided by compiling a source program using a vector processing option. Thus, as the implementation is not directly coded at the cache level, there is likely to be a discrepancy between the predictive model above and the actual implementation. For example, further refinements of the implementation model are possible: the black vector $\mathbf{u}_{i-1,j_C}$ could be stored, used and subsequently shifted so that a quad word aligned version of the vector $\mathbf{u}_{i+1,j_C}$ is generated.

Appendix C: A model of the block implementation

The implementation of a block red phase smoothing operation is outlined below:

TRANSFER a black block of **u**	from MAIN MEMORY to CACHE;
TRANSFER a red block of **f**	from MAIN MEMORY to CACHE;
CALCULATE a red block of **u**;	
TRANSFER the red block of **u**	from CACHE to MAIN MEMORY.

u is represented in main memory as a 2D array. A subpattern of black **u** points which are to be transferred to cache are pictured below:

```
R  X  R  X  R
X  R  X  R  X
R  X  R  X  R
X  R  X  R  X
R  X  R  X  R
```

where X is a black point. In order to conform to the software restrictions for transfer to cache (Appendix A) the black points are copied in two phases: first the black points in odd columns and then the black points in even columns. A similar strategy (in reverse) is used to transfer the red **u** points from cache to main memory. In effect this means that the **u** data is four coloured in cache (two shades of red and two shades of black). The cache memory is used to store four (12 × 12)-subblocks of **u** - one for each colour - and two (10 × 10)-subblocks of **f**. (Again a transfer restriction prohibits the use of (11 × 11)-subblocks.) A new approximation of red values is calculated for the two red-coloured (10 × 10)-subblocks. The method has been implemented using scalar arithmetic for three reasons:

(i) pipelined processing is only worthwhile for (10 × 10)-blocks if long vector processing is applied. This is problematic for reduced sized boundary blocks;

(ii) the data needs to be shifted internally in cache in order to achieve suitable quad word aligning;

(iii) there are no library subroutines for (cache) DOUBLE PRECISION vector - vector addition.

If $n_{1/2}$ for loading/storing equals 50 then the average time to load an element of a (12 × 12)-block is 67 nanoseconds and the average time to load/store an element of a (10 × 10)-block is 75 nanoseconds. These adjusted loading and storing times may be used to predict the performance of the implementation.

Appendix D: Model 1 of the "operation composition" implementation

In order to prevent unwanted interference it is necessary to use different arrays to store the input **u** data and the output **u** data. The strategies for loading and storing data to/from cache are similar to those used in the **block** loaded implementation of Appendix C. Thus, **u** subgrids are again four coloured in cache. Cache is organised into four blocks of size 12 × 12 - in order to represent a **u** subgrid of size 22 × 22 - and four blocks of size 10 × 10 - in order to represent an **f** subgrid of size 20 × 20. The residuals are calculated for four (8 × 8)-blocks - a subgrid of size 16 × 16. Scalar arithmetic is used for the implementation (as in Appendix C).

The predicted performance of this implementation is calculated using $n_{1/2}$ for loading/storing to be 50. This means that the estimated time to load an element of a (12 × 12)-subblock is 67 nanoseconds, load an element of a (10 × 10)-subblock is 75 nanoseconds, load an element of a 12-vector is 258 nanoseconds and store an element of an (8 × 8)-subblock is 89 nanoseconds. A residual calculation is defined by:

$$\text{TERM} := (\mathbf{u}_{i+1,j} + \mathbf{u}_{i-1,j} + \mathbf{u}_{i,j+1} + \mathbf{u}_{i,j-1} - 4\,\mathbf{u}_{i,j} - h^2\,\mathbf{f}_{i,j})$$

$$\text{SUM} := \text{SUM} + (\text{TERM} \times \text{TERM})$$

i.e. 6 additions and 2 multiplications. The details of the predicted time (nanoseconds) to compute four (8×8)-blocks of results are listed below:

$2\ \{(12 \times 12) \times 67\}$	LOAD black blocks of **u** points
$4\ \{(10 \times 10) \times 75\}$	LOAD all blocks of **f** points
$2\ \{(12 \times 1) \times 258\}$	LOAD boundary red points (worst case)
$2\ \{(10 \times 10) \times 400\}$	COMPUTE new approximations for red blocks
$2\ \{(9 \times 9) \times 400\}$	COMPUTE new approximations for black blocks
$2\ \{8 \times 8) \times 650\}$	COMPUTE new residuals for red blocks
$4\ \{(8 \times 8) \times 89\}$	STORE all blocks of **u** points

Therefore, 2304 (8*8*2)(5+5+8) arithmetic operations take 306272 nanoseconds. This is equivalent to a rate of 7.5 MFLOPs per processor with a leverage of 2.9.

Appendix E: Model 2 of the "operation composition" implementation

The predicted performance of this implementation is calculated using $n_{1/2}$ for loading/storing to be 50. This means that the estimated time to load an element of a (20×20)-subblock is 56 nanoseconds, load an element of a (18×18)-subblock is 58 nanoseconds and store an element of an (14×14)-subblock is 63 nanoseconds. The details of the predicted time (nanoseconds) to compute a (14×14)-blocks of results are listed below:

$(20 \times 20) \times 56$	LOAD **u** points
$(18 \times 18) \times 58$	LOAD **f** points
$0.5\ (18 \times 18) \times 400$	COMPUTE new approximations for red points
$0.5\ (16 \times 16) \times 400$	COMPUTE new approximations for black points
$0.5\ (14 \times 14) \times 650$	COMPUTE new residuals for red points
$(14 \times 14) \times 63$	STORE **u** points

Therefore, 1764 arithmetic operations take 233240 nanoseconds. This is equivalent to a rate of 7.6 MFLOPs per processor with a leverage of 3.4.

References

[1] *Cray APP System Overview,* Cray Research Superservers, Inc., 1992.

[2] C.-E. Fröberg, *Introduction to Numerical Analysis,* Addison-Wesley, 1965.

[3] W. Hackbusch, *Multi-grid Methods and Applications*, Springer Ser. Comput. Math. 4, Springer, Berlin, 1985.

[4] R.W. Hockney, *Characterizing overheads on VM-EPEX and multiple FPS-164 processors,* Parallel Systems and Computation (edts G.Paul and G.S. Almasi), North Holland, 1988, 255-271.

[5] R.W. Hockney and I.J. Curington, $f_{1/2}$: *A parameter to characterize memory and communication bottlenecks*, Parallel Computing 10, 1989, 277-286.

[6] *Intel, i860 64-bit microprocessor programmer's reference manual,* Intel 1989.

[7] *Kendall Square Research*, Waltham, Ma., KSR1 Principles of Operation, 1991.

[8] C.H. Lai, H.J.J. te Riele and A.Ualit, *Parallel experiments with simple linear algebra operations on a Cray S-MP system 500 matrix coprocessor*, CWI Note NM-N9301, 1993.

[9] L.G. Valiant, *A bridging model for parallel computation*, CACM 33, (8), 1990, 103-111.

Acknowledgements:

Alan Stewart acknowledges the financial support provided by the ERCIM Fellowship Programme and the Commission of the European Communities.